Joao Luis Silva Damas
Geoff Huston
September 2016

Binding to an IPv6 Subnet

In the original framework of the IP architecture, hosts had network interfaces, and network interfaces had single IP addresses. The list of active network interfaces, and the manner in which they acquire IP addresses, either by a static configuration or by some dynamic mechanism via a query to the local network, is specified in a boot time configuration file.

For example, on a Debian Linux host system the configuration file `/etc/network/interfaces` enumerates the host's active network interfaces and the same in which they are configured with an address. For example, one might see in that file some elements to configure the interface `eth0:`

```
iface eth0 inet static
    address 198.51.100.2/24
    gateway 198.51.100.1

iface eth0 inet6 static
    address 2001:db8::2/64
    gateway 2001:db8::1
```

In this example the `eth0` interface is configured statically with a single IPv4 address and a single IPv6 address, together with a protocol-specific default route (the '`gateway`' element) to allow the host to know where to send non-locally addressed packets.

While the name and location of the network interface configuration file may change across various flavours of Linux, and will change across different operating systems, the same basic boot-time network interface configuration instructions are present on most systems.

These days, many operating systems allow a configuration to add additional addresses to network interfaces. This "aliasing" of an interface with additional IP addresses allows a host to treat a number of differently IP addressed packets as locally-address packets and allows applications on the host to bind to all of these addresses simultaneously.

Continuing with the Debian example, one way of configuring this is to duplicate the interface descriptions with these alternate addresses:

```
iface eth0 inet6 static
    address 2001:db8::2/64
    gateway 2001:db8::1
```

```
iface eth0 inet6 static
    address 2001:db8:10::10/64
```

(There are a number of ways to configure these alias addresses in a Debian host, and this example is just one way, but let's not head down that path here!)

Multiple interface addresses have also been adopted in the IPv6 architecture, where network interfaces can have multiple addresses, including scoped addresses (IPv6 link local addresses), dynamic addresses (IPv6 privacy addresses) and multiple globally scoped addresses (multi-homed site prefix addresses).

However, in all these cases we are talking about binding an interface to an enumerated list of individual host addresses.

Even this simple enumeration of alias addresses across a number of network interfaces can present some challenges to application. A good explanation of the issues here, and an approach to a solution can be found in a blog entry from the PowerDNS folk. https://blog.powerdns.com/2012/10/08/on-binding-datagram-udp-sockets-to-the-any-addresses/ considers the API coding required to bind to all local addresses from a server application.

This enumeration approach has its limitations in terms of the number of addresses that can be managed in this fashion. What if, for example, in an IPv6 context you wanted the host to treat an entire /64 subnet as a collection of local addresses that are all equally concurrently available to the host? In other words, what if you wanted an application to be able to receive packets that were addressed to any 128 bit address within this /64 subnet? Obviously enumeration is not viable with such a large set of addresses so we have to look elsewhere for a solution.

To achieve this outcome, we could resort to installing some form of local address translation functionality, and use a translation rule that transforms all the addresses in the subnet to a single local address. By placing this rule into a host's internal firewall/packet filter engine all packets addressed to the subnet could be mapped into a single host address on ingress, and presumably also mapped back on egress. However, this approach has its limitations. Address translation, even if performed within the host, will not expose the original host address to the application. So if we would like the application to not only respond to packets addressed to any host address within a subnet, but also to be aware of which address it is responding to then we need to find a method that directs the host to accept all packets addressed to the subnet as "locally addressed' packets, and pass them to applications as necessary without changing the address values in the packet.

Another way of phrasing this question is: How can we bind a network interface to an entire subnet of IP addresses without having to enumerate each and every individual address?

The Linux *local* Routing Table

In the same way that host operating systems have evolved to managed interfaces with multiple IP addresses, we have also evolved our thinking about the internal packet forwarding structure in hosts. The original model of a host was a single routing table, that specified how to handle packets. In its simplest form each entry in the routing table specified an address prefix and an interface, so that a packet that is processed by the routing subsystem is directed to the nominated host interface according to the lookup of the packet's destination address into the routing table.

However, this view of the local routing structure has been revised by more recent versions of common operating systems. For example, the Linux kernel version 2.2 and version 2.4 support up to 254 routing tables. The original intent of this innovation was to allow flexibility in packet handling so that some local route policy setting could direct the host to use a different route table other than the default packet handling rules as defined by the *main* routing table. In these Linux systems two of these routing tables always exist: the *local* routing table (id 255) and the *main* routing table (id 254). The *main* routing table is the conventional one that describes the default local packet handling regime. The *local* routing table is normally controlled by the kernel to keep track of local (to the machine) IP addresses. For example, when additional IP addresses are added to a local interface, a host address unicast route entry is automatically added to the *local* route table. In this way the operating system can track which addresses are associated with which network interface.

As well as mapping interface alias IP addresses into the *local* route table, it is possible to enter a route into this table without explicitly configuring a local interface. in this case if a route is entered into the *local* routing table pointing to the local loopback interface, then this is directing the local kernel to regard any matching packet as one that is directed to itself, rather than to be forwarded onward. Interestingly this functionality in the *local* routing table applies to both unicast host routes and unicast prefix routes.

```
> ip route add local 2001:db8:1:1::/64 dev lo
```

We can also list the contents of this *local* routing table:

```
> ip -6 route list table local
local ::1 dev lo  proto kernel  metric 256
local ::1 dev lo  proto none  metric 0
local 2001:db8::2 dev lo proto none  metric 0
local 2001:db8:1:1::/64 dev lo  metric 1024
local fe80::f03c:91ff:feb0:ffff dev lo  proto none  metric 0
ff00::/8 dev eth0  metric 256
```

What we see is the /128 address binding from an explicitly configured interface address (the entry for the host address `2001:db8::2`) and the explicitly locally routed /64 subnet `2001:db8:1:1::/64`.

This implies that to enable binding a local process to a full prefix of addresses without explicitly configuring each of them on an interface, we can inject a route for the prefix pointing to loopback into the local routing table and from then on the kernel will behave as if all those addresses are configured on the machine.

This is of course not all that needs to be done – this configuration step just allows a host system to recognize that incoming packets that are directed to a network interface are classified as "locally addressed" packets. The local router also needs to have an explicit route entry for the subnet that directs all packets with destination addresses that are in this subnet to this host.

Using a Subnet Binding

For TCP-based services this *local* route table subnet entry is sufficient to allow a local application to listen and respond to incoming connections on an entire subnet. The component of TCP API that can bind to locally configured subnets is the `bind` socket call, that binds the socket to an interface or a set of IP addresses. The API variant we need to use to perform the subnet binding is to pause the generic wildcard address (textually this is equivalent to binding the socket to the 0.0.0.0 IPv4 address and to the :: IPv6 address) . Given that each TCP session keeps session state, the replies that the TCP server generates uses a source address that matches the destination address of the incoming packet. In this manner the TCP server's socket can be bound to all local addresses, including all addresses defined within these locally routed subnets simply by binding the listening socket to the wildcard IP address,

and the expected thing will happen: namely the TCP server will respond to connection requests addressed to any host address in the locally defined subnet range.

This is not sufficient in the case of a UDP-based server wanting to bind to a subnet configured by this *local* routing table approach. As with a TCP server, a UDP server must bind to the wildcard address so that it will accept any addressed packet that is accepted on any local interface. However, there is no saved session state at play here, so while the wildcard binding is sufficient to allow a UDP server gather all incoming packets directed to the relevant UDP service for packets addressed to any address within the entire subnet, any response that the UDP server generates will not use the original destination address as its source address. What it will use is the IP address of the routing-determined outbound network interface for the packet. Obviously this can present problems to a UDP client, particularly when it is using a UDP 4-tuple of the source and destination addresses and port numbers in order to match responses to queries.

What we would strongly prefer is that the address pairing is preserved, such that the source address of a UDP server's response is the same as the destination address used to contact the server in the first place. To achieve this, the UDP server application needs to set the source address or outgoing responses explicitly, using the destination address gathered from the incoming UDP packet it is responding to. Picking up the destination address from the socket API is a case of adding a pointer to a message control information block to the socket `recvmsg()` call that collects an incoming UDP message and passes it to the UDP server. There are a number of ways to do this in IPv4, depending in the operating system, but in the case of IPv6 this has been standardized in RFC3542. We need to set the `IPV6_RECVPKTINFO` parameter on the socket. We can then call `recvmsg()` with an associated message control information block. The block that has the type value `IPV6_PKTINFO` has an associated data block that contains the destination IPv6 address of the UDP packet, and the local host's interface index that received the packet.

So now that we have the original destination address from the incoming packet we'd like to set it as the source address of the outgoing packet. RFC3542 specifies the same API approach to perform this, where the control block with type `IPV6_PKTINFO` has a data block that contains the source IPv6 address for the outgoing packet, in the same format as the control block used to receive the packet.

Now we are almost there. If we set the source address to any host address associated with a network interface than that's all we need to do. But our subnet binding is not a host address, and efforts to use an address drawn from this subnet will conventionally fail a check in the socket driver. What we need to do is direct the socket to operate in a more promiscuous mode and accept any of these subnet addresses as source addresses for outgoing packets. In the case of Debian systems the socket option `IP_FREEBIND` is required to allow the UDP server to respond to using a source address that matches the incoming UDP packet's destination address when using this subnet address space.

At this point we are done (at least for Debian)! We can associate an entire subnet to the host, and as long as the external environment directs packets addresses to any address within the subnet to this host we can respond in a conventional manner in both TCP and UDP.

> Source code for both TCP and UDP servers for Debian Linux (version 8.4 of Debian) is located at <http://www.potaroo.net/ispcol/2016-09/>. Also we've included the source code for a generic test server. Perhaps its best left as a challenge to an interested reader to uncover if there are comparable settings that allow this subnet binding functionality in other systems, such as FreeBSD and Mac OSX. Please let us know what you find.

So, in conclusion, yes, it is possible to construct in both TCP and UDP a host system that allows applications to bind to entire subnets, and respond to incoming traffic directed to any address drawn from the subnet, and to do so efficiently without performing address translation or any other form of packet transformations, and without any overhead of extensive local table lookups that would otherwise be the case with large enumerated address lists.

Where is this subnet binding technique useful? We will describe our application of this technique in the next article.

Authors

Joao Luis Silva Damas is currently Senior Researcher at APNIC. Joao was also co-founder of Hivecast Inc, a DNS services company later sold to Dyn. Previously he worked at ISC (Internet Systems Consortium) as CTO overseeing technical developments. Earlier, he served as CTO at RIPE NCC and later founded Bond Internet Systems as consulting and research company. For around 7 years he organised the RIPE plenary program and launched the current RIPE program committee. In 2008, together with colleagues, launched ESNOG to bring together Spanish ISPs to interact with each other, an activity that continues to this day.

Geoff Huston B.Sc., M.Sc., is the Chief Scientist at APNIC, the Regional Internet Registry serving the Asia Pacific region. He has been closely involved with the development of the Internet for many years, particularly within Australia, where he was responsible for building the Internet within the Australian academic and research sector in the early 1990's. He is author of a number of Internet-related books, and was a member of the Internet Architecture Board from 1999 until 2005, and served on the Board of Trustees of the Internet Society from 1992 until 2001 and chaired a number of IETF Working Groups. He has worked as an Internet researcher, as an ISP systems architect and a network operator at various times.

www.potaroo.net

Disclaimer

The above views do not necessarily represent the views or positions of the Asia Pacific Network Information Centre.